ECE 4750 Lab 3: Blocking Cache

Peter Fanning Wade (pfw44), Basant Amro Khalil (bak94), and Lakshmi Aparna Bolla (lb632)

November 15, 2021

Section 1. Introduction

One of the main factors affecting processor performance is memory latency. Accessing main memory can take hundreds of cycles, bottlenecking the processor. One way to reduce the effect of this memory delay is to incorporate a cache into the architecture: a smaller, fast memory, usually SRAM, that holds instructions and/or data that the processor can access without having to wait for a response from the main memory. Caches are faster than main memory because they are smaller, are located close to the processor, and are made in a way that trades size per byte for better performance. A cache hit occurs when the data we are requesting is in the cache, while a cache miss occurs when the data we are requesting is not in the cache and thus requires accessing main memory. Caches exploit spatial and temporal locality to increase the number of cache hits. In an address sequence with a significant spatial locality, we access memory addresses that are close together and often contiguous. In an address sequence with a significant temporal locality, we often access the same addresses multiple times. The lab incorporates many important themes in computer architecture like analyzing cache performance through hit and miss rates, as discussed in lecture. We designed a control unit and datapath with a split design pattern, building on what we learned in labs 1 and 2. We used an incremental development design and testing methodology by beginning with a baseline design before moving to an alternate design. In this lab, we will design two different cache architectures. The baseline design is a direct-mapped, write-back, write-allocate cache. The alternative design is a two-way set-associative cache that should reduce the miss rate by avoiding conflict misses, both of 256B capacity. Both caches used a write-back, write-allocate for handling write misses, meaning that writes are only written to memory when that cache line needs to be evicted from the cache, rather than writing through to memory every time we write to the cache. In a direct-mapped cache, locations in memory map to exactly one location in the cache, but in a two-way set-associative cache, locations in memory can map to one of two different ways. This would lead to fewer conflict misses with the set-associative cache since multiple memory locations with the same index bits can be mapped to the cache simultaneously, which leads to improvement in the Average Memory Access Latency (AMAL) over a direct-mapped cache. Thus, we expect an improvement in the AMAL for our two-way set-associative cache design, which is the alternative design, over the direct-mapped cache design, the baseline design. The alternative design will have a similar design as that of the baseline design except that the control signals will be a bit different as well as the datapath with some additional muxes. For instance, we split the valid bits into two parts, one for each way. We also AND the result of the tag match in each way with the appropriate valid bit to determine if there is a hit or miss as there could be a hit one of the ways. If it is a miss, then it should be a miss in both ways. The control unit in the alternative design also used LRU(Least Recently Used Policy) to choose between the two ways during eviction when needed. For both designs, we created components in the datapath, set values in the control unit, and edited the parent module, connecting the components into the final designs. In this lab, we implemented the baseline and alternative designs, verified the designs using an effective testing strategy, and performed an evaluation comparing the two implementations. The alternative design will likely use more power than the baseline design since more comparisons are done, more registers are read at a given time, and additional muxes. The alternative design will also require a larger area since additional logic is required for the address decoding for two smaller register files compared to a single larger register file. The muxes will also contribute to the additional space. The least recently used register file is going to take more space as well. These changes will also slow down clock frequency as additional muxes are on the critical path increasing the minimum cycle time. In addition, the logic to write and read from the registers is more complex, and, as a result, the alternative design will be slower than that of the baseline implementation. We compared both designs over some simulated memory access scenarios. For a pattern that was looping over the same array, exhibiting both spatial and temporal locality, but the time between the repeated accesses were spaced out, so the temporal locality was not exploited, the baseline design performed better than the alternative design with an amal of 8.464 for the baseline design versus 9.752 for the alternative design. For the 3d loop with temporal and spatial locality, the alternative design performed better than the alternative design with an amal of 8.6125 compared to the amal of 27.0125 for the baseline design. For other designs with temporal and spatial locality, the alternative design performed better than the baseline design, such as in vv_add_tmp and vv_add_spac. As such, we can conclude that, in some cases, the alternative design might be a better implementation than the baseline design providing better amal and being more power-efficient as a whole, despite using more power per cycle. Also, even though the clock speed of the alternative design might be slower, the implementation of the alternative design and processor might be faster if the lower AMAL of the alternative design means the processor can reduce the stalling needed for memory accesses. As such, the alternative design would be better to use in a full system, as the disadvantages from the more complex design are negated by the increased hit rate, which would overall mean that the processor-cache system has to waste fewer cycles fetching data and therefore requires less time and energy overall. The alternative design is more compelling than the baseline design for larger caches because it can reduce the miss rate by reducing the number of conflict misses. This is due to the fact that the baseline (direct-mapped) design only allows cache lines to be placed in a single location, whereas the alternative (two-way set-associative) design allows cache lines to be placed in one of two locations. In general, increasing the number of ways in a set-associative cache reduces the number of conflict misses and, in turn, reduces the miss penalty. Direct-mapped caches, on the other hand, optimize hit latency and area/power requirements for smaller caches, such as those directly below the processor level.

Section 2. Baseline Design

The baseline design is a direct-mapped, write-back, write-allocate cache with a total capacity of 256 bytes, 16 cache lines, and 16 bytes per cache. The baseline design was split into control, datapath, and top-level modules to make design and debugging easier but also to enforce the design principles of modularity, hierarchy, and encapsulation for a design of this level of complexity. The baseline design utilizes the control/datapath split design, and the control unit uses an FSM. This decomposition is an example of the modular design principle, which is an approach that subdivides a system into smaller modules that can be independently created. This method is utilized because it allows the different parts to be independently created and used in different systems, which is effective as well as efficient as multiple copies of the same sub-system do not need to be created in different systems. The cache control is managed by a finite state machine controller (FSM) that deals with states with one or more steps per state during which control signals are fed out to the datapath and status signals are received in return. State machine and datapath diagrams for the baseline design are shown in Figures 1 and 2. We utilize val/ready interfaces for instruction/data memory requests which allows the cache to operate independently of the memory latency. The interface for the direct-mapped cache consists of the following inputs: clock and reset signals, the test source/cache request message and valid signal, cache response ready signal (from the test sink/processor), memory request ready signal, and the memory response message and valid signal. The outputs are the cache request ready signal, cache response message and valid signal, memory request message and valid signal, as well as the memory response ready signal. We began working on the design by first fully creating the datapath and then adding different paths: init transaction, read hit path, write hit path, refill path, and evict path. Since the cache is write-back, write-allocate, we write only to the cache on a write hit and load the cache line from memory (and then write to the cache) on a write miss. Also, we maintain two register files in the control module for keeping track of valid and dirty cache lines. This information is necessary for determining whether evictions are required on cache read/write misses.

In our datapath, we had a comparator to figure if we got a tag match. We also added a write data mux to select which word we need to write to and a read word mux to select which word to read from. The main components of datapath were the tag and data arrays. Other components were registers and muxes, which were needed for correctly storing and moving information into and out of the arrays. The baseline datapath, as shown in Figure 1, was implemented all at once since it was relatively simple and most paths needed to be in place for basic transactions. To handle valid and dirty bits associated with the cache lines of our datapath, we implemented register files in our control unit. One register file was for valid bits; the other register file was for dirty bits.

Our datapath exhibits the principles of modularity and encapsulation as the tag and data SRAMs, along with the valid and dirty register files, did not affect the rest of the datapath - they could simply be inserted as self-governing modules. In terms of our control path, the FSM diagram is shown in figure 2 below. We implemented 14 states to capture all conditions. We had Idle state (I), Tag Check (TC), Evict Prepare (EP), Evict Request (ER), Evict Wait (EW), Refill Request (RR), Refill Wait(RW), Refill Update(RU), Write Data Access Miss (WDAM), Write Data Access Hit (WDAH), Read Data Access Miss (RDAM), Read Data Access Hit (RDAH), Init Data Access (IN), Wait Miss (WM) and Wait Hit (WH). The idle state receives the incoming cache request, places it in the input registers, and moves to the TC state once the cache request is valid, otherwise, it stays in the idle state. The TC state checks the tag and moves to the RDAH state on a read hit, the WDAH state on a write hit, IN state on an initial transaction, the EP state on a miss and a dirty cache line, and the RR state on a miss and a non-dirty cache line. The IN state immediately writes to the appropriate cache line, and then the cache moves to the W state. The RD state reads from the appropriate cache line, and then the cache moves to the W state. The WD state writes to the appropriate cache line, and then the cache moves to the WM state. The WH and WM states wait for the cache response to be ready and, once it is, the cache moves to the I state. The EP state reads the tag and data, prepares the eviction message, and then the cache moves to the ER state. The ER state makes a request to memory to write the evicted cache line and moves to the EW state once the memory request is ready. The EW state waits for the memory response and moves to the RR state once the memory response is valid. The RR state makes a request to memory to write the evicted cache line and moves to the RW state once the memory request is ready. The RW state waits for memory response and moves to the RU state once the memory response is valid. The RU state writes the response to the victim cache line and moves to the RDAM state if the type is a read or to the WDAM state if the type is a write.

This design is a good baseline for comparison because a direct-mapped cache is a basic cache without optimizations that try to reduce the miss rate. These optimizations require greater complexity in our control logic, as well as more hardware in our datapath. Therefore, this design provides a good baseline to build on and optimize later in our alternative design in order to improve the hit rate. Being able to compare other designs to a direct-mapped cache allows us to easily evaluate other designs and draw conclusions about the hit and miss rates. Overall, the results of the evaluation demonstrate the differences between a direct-mapped and a two-way set-associative cache. A direct-mapped cache, much like our baseline design, only allows cache lines to be put in one location. Therefore, the design is not optimized for hit rate because the baseline design misses the opportunity to take full advantage of the cache's capacity. However, as previously discussed, the alternative design optimizes both the hit rate and energy consumption through increased performance and efficiency, though possibly at the cost of higher latency. Although the area consumption was not decreased with the alternative design compared to the baseline design, the upgrades that the alternative design brings are far more important for supporting higher cache levels than the amount of area that it consumes. Therefore, we can conclude that the alternative design is better optimized for situations where the hit rate is critical, such as the larger caches closer to the main memory, while the baseline design is better for small caches closer to the processor.

Section 3. Alternative Design

For our alternative design implementation, we implemented a two-way set-associative cache. We used a very similar FSM to the baseline design, adding no new states and only a few new control signals. Compared to the direct-mapped cache, the set-associative cache results in a higher hit rate since, for any given index, there are two different ways where the tag can be stored. At the same time, however, associativity comes at the cost of additional hardware requirements, more complex control logic, and a potentially greater latency. Specifically, a set-associative design requires separate SRAM modules for the way zero and way one tag arrays, and it requires control and status signals to determine which of the two ways produces a hit. Additionally, in the case of a miss, there needs to be a replacement policy to decide which of the two ways to overwrite. In this case, we implement a least-recently-used replacement (LRU) policy to choose between the two ways during eviction as this policy is empirically demonstrated to have good performance. Like the baseline design, the alternative design is based on the datapath/control split principle, where the control unit sends signals to the datapath according to the state of the FSM logic. The control logic for the alternative design is mostly the same as that of the baseline design, as we use the exact same states; the only difference is that there are a few new signals that help control which way to read or write to. As such, the FSM for the alternative design can also be seen in Figure 1. We chose to do this as most of the signals required to determine which way to use were already in the datapath, and adding states to differentiate actions for way zero or way one would add too much complexity to the FSM. We split up the tag and data arrays into two, each with eight lines. We did the same for the valid and dirty registers. We added new registers to keep track of the least recently used ways, so we could properly implement the LRU replacement policy. We added multiplexers to determine which way output to use. This new datapath can be seen in Figure 3. We kept the valid/ready interface from the baseline design. The most complicated part was determining when to write or read from each way, as well as what way to overwrite. To do this, we used the bits indicating that the entries were valid, as well as tag match bits and the least recently used register output. We were able to reuse much of the baseline design because of the modularity. We could simply adjust the size of the modules, such as the tag and data arrays, and then use the components.

A two-way set-associative design such as the one we implemented reduces the number of conflict misses over a direct-mapped cache. Since data with the same id bits is not always removed, that data can remain in the cache to be used again. This is advantageous for exploiting temporal locality, as the LRU replacement policy will keep data that is frequently used. The two designs should behave similarly on spatial locality, as they have the same sized cache lines. We chose to only do a two-way set-associative cache instead of a fully associative cache, as fully associative caches require a much more complex control system to determine which line to use, and this additional overhead may not be worthwhile for caches that are not super small. We chose 16 cache lines, each of which stores 16 B of data, as this provided a good balance between the size of each line, which allows for better exploitation of spatial locality, and enough lines that they are not replaced too frequently, which aids in exploiting temporal locality.

Section 4. Testing Strategy

We used incremental and test-driven methodologies for completing both the baseline and the alternative design. That is, we verified our implementation of each instruction (read hit, write hit, read miss, and write miss) before continuing to the next one by running the provided basic test and a few simple directed value tests we write during the development process. After fully implementing the datapath and control modules, we comprehensively tested each instruction by adding directed test cases for different cache paths. We also added random value testing as well as random delay testing to ensure that the processor could handle all cases and the uncertainty of real-world computation. We combined these directed and random tests into unit test suites for each instruction that not only verified the correctness of the processor in isolated tests but also in sequences of instructions. Throughout the testing process, we use the line trace outputs of the unit tests to locate errors and debug faulty connections and control logic in our modules. For both designs, it was important to test transactions that would hit/miss based on the cache microarchitecture. We wrote directed tests that specifically target different functions of the datapath and FSM. For example, we tested filling an entire cache line, conflict misses for the same line of the cache, reading/writing to all four words in the cache line, writing to every line of the cache, and capacity misses. To ensure the correctness of these cases, we tailor the index bits of the addresses to create hits, misses, and evictions. For the alternative design, we wrote tests to check that the way selection was properly implemented. For example, we had to test writing to the same set in both ways, conflict misses in both ways, filling the entire cache (which uses different addresses than for direct-mapped design), and evicting/refilling the correct way of the cache. We particularly wanted to test writing to the ways in patterns other than a purely alternating pattern by writing to the same address multiple times in a row. Directed tests were very useful in debugging our code as well as ensuring that we could handle edge cases correctly. To make sure that we handled the ready and valid interfaces correctly, we added random stalls and delays to the source, sink, and memory. These tests ensure that our cache will perform correctly in the uncertain environment of real computation. These directed tests allowed us to ensure that our designs were able to correctly process a variety of corner cases.

We also wrote fully randomized tests that simulate cache operation in general situations. We randomly determined the addresses and values for each location. To ensure that we were able to properly determine if the caches hit or missed, we simulated the tag arrays in python while generating the test cases. We tested a variety of situations, such as random addresses and values read-only, combining reads and writes, and strides with random values. These random tests allowed us to feel confident that our caches are correct in a general case.

Section 5. Evaluation

We ran our two implementations through a variety of memory access patterns and observed the hit and miss rates. The data can be seen in Table 1. The first pattern was a simple search through a one-dimensional array. This pattern demonstrates spatial locality but not temporal locality. Both designs performed the same, as all the misses were compulsory misses. The next pattern was repeatedly looping over the same array. This pattern includes both spatial and temporal locality, but the temporal locality is not exploited much as the time between the repeat accesses is too large. The baseline implementation actually performed better than the alternative design for this pattern, as it likely was able to exploit temporal locality to a small degree due to the increased number of indexes. The next pattern was an iteration through an array in a three-dimensional pattern. Again this pattern has both temporal and spatial locality. The alternative design performed better, as it was able to store multiple entries with the same id bits, while the baseline design would evict them from the cache.

Our next pattern was vv_add_tmp. This design is intended to simulate the memory access pattern of adding multiple vectors together. The pattern was designed to best exploit the temporal locality of this pattern by doing the additions one row at a time. The alternative design performed best, as it was again able to exploit temporal locality while the baseline design could not. The next pattern was vv_add_spac. This is similar to the previous one, except that we do the addition of the entire vector before we move to the next one. This better exploits the spatial locality of the pattern. Again, the alternative design performed better. Our next pattern was randomly generated. We select random memory locations, and as such, we would not expect any sort of spatial locality. We do access each location multiple times, so there is some temporal locality. The baseline design performs best for this pattern, but both designs have a significant number of misses. Our next pattern was reverse_arr. This is designed to simulate reversing the order of an array. This pattern has both spatial and temporal locality, as we read and write the same locations multiple times, and all the locations are near each other in memory. The baseline design performed better on this, although both behaved very similarly.

Our next pattern was lin_search. This is designed to simulate the access patterns from performing a linear search through an array for a value. This pattern has both spatial and temporal locality, as most of the items are in sequence in memory, and there are repeated accesses to the “target” value. The alternate design was better able to exploit this locality and performed better. Our next pattern was read_wr_repeat. This pattern repeatedly reads and writes a set of memory locations. This pattern was specifically designed to work well with our alternative design, as the memory addresses have eight different id bit patterns and two different tags. As such, the alternative design performed far better than the baseline design, missing only one-thousandth the number of times as the baseline. Our next pattern was read_miss. This design has no temporal or spatial locality, and as such, both the baseline and alternate designs perform identically poorly. The next pattern is read_miss_repeat. This adds some temporal locality to the previous pattern, as the locations are accessed multiple times. The alternative design performs a bit better than the baseline but not substantially. The final two patterns, write_miss and write_miss_repeat, are the same as above except replacing the reads with writes. The miss patterns are, therefore, the same as the read versions, but the average memory access latency is different, as the cache has to evict the dirty lines.

The alternative implementation will likely use more power than the baseline design. Since more registers are read at a given time and more comparisons are done. The additional muxes will also contribute to this increased power usage, as additional logic must be used. In addition to increased power consumption, the alternate design will require more space. Even though we cut the size of the registers in half, additional logic is required for the address decoding for two smaller register files compared to a single larger register file. The muxes will also take up additional space, especially as one of the signals is large, 128 bits. We also need more space for the least recently used register file, as well as the control logic in selecting the correct way. These changes will not only increase the size but also slow down clock frequency. Additional muxes are on the critical path, increasing the minimum cycle time. The logic for reading and writing to the registers is also more complex, and therefore slower than that of the baseline design. This decrease in speed will not counteract the increase in energy consumption per cycle since there are so many more components that are in use at a given time.

In some cases, however, the alternate design may be more power-efficient than the baseline. Since cache misses require many cycles to access main memory and refill the cache, in situations where the average memory access latency for the alternate design is less than that of the baseline design, the alternate design may be more power-efficient. Similarly, while the clock speed of the alternative design is likely slower, the full implementation of the alternate design and a processor may be faster if the lower AMAL of the alternate design means the processor can reduce the amount of stalling required for memory accesses. As such, the alternate design would be better to use in a full system, as the disadvantages from the more complex design are negated by the increased hit rate, which would overall mean that the processor-cache system has to waste fewer cycles fetching data and therefore requires less time and energy overall.

Section 6. Role and Task Table

Member 1: Peter Wade	**Member 2: Lakshmi Aparna Bolla**	**Member 3: Basant Khalil**
Verification Writing Test cases and debugging, along with the other members, the errors that showed up until the code for the baseline design passed the test cases. Providing some suggestions to fix some of the errors that existed in the code. Contributing to the lab report through contributing to sections, such as evaluation, baseline design and alternative design sections. Contributing to the baseline datapath and control module logic Coming up with the alternative block diagram. Contributing to the alternative datapath and control module logic Providing suggestions of the tests to be added and adding them	RTL+architect Helping with debugging some bugs in the main baseline code and contributing to the datapath diagram for the alternative design. Helping write the logic for the control module in the alternative design and contributing to the control module Helping create test cases for instructions. Providing suggestions to other team members along the way of fixing the errors showing up. Helping with writing the code for the baseline design and providing suggestions to help with debugging the design when we faced some errors. Contributing to the lab report through contributing to sections like referenced figures, introduction and alternative design.	RTL Contributing to sections in the lab report, such as the introduction, baseline, and alternative design sections. Helping contribute with some code logic for the baseline data path module and the control module Coming up with the baseline design FSM diagram Providing some suggestions for the tests to be added Working with team members for the code debugging issues Helping contribute to the alternative design control and datapath modules

Section 7. Referenced Figures

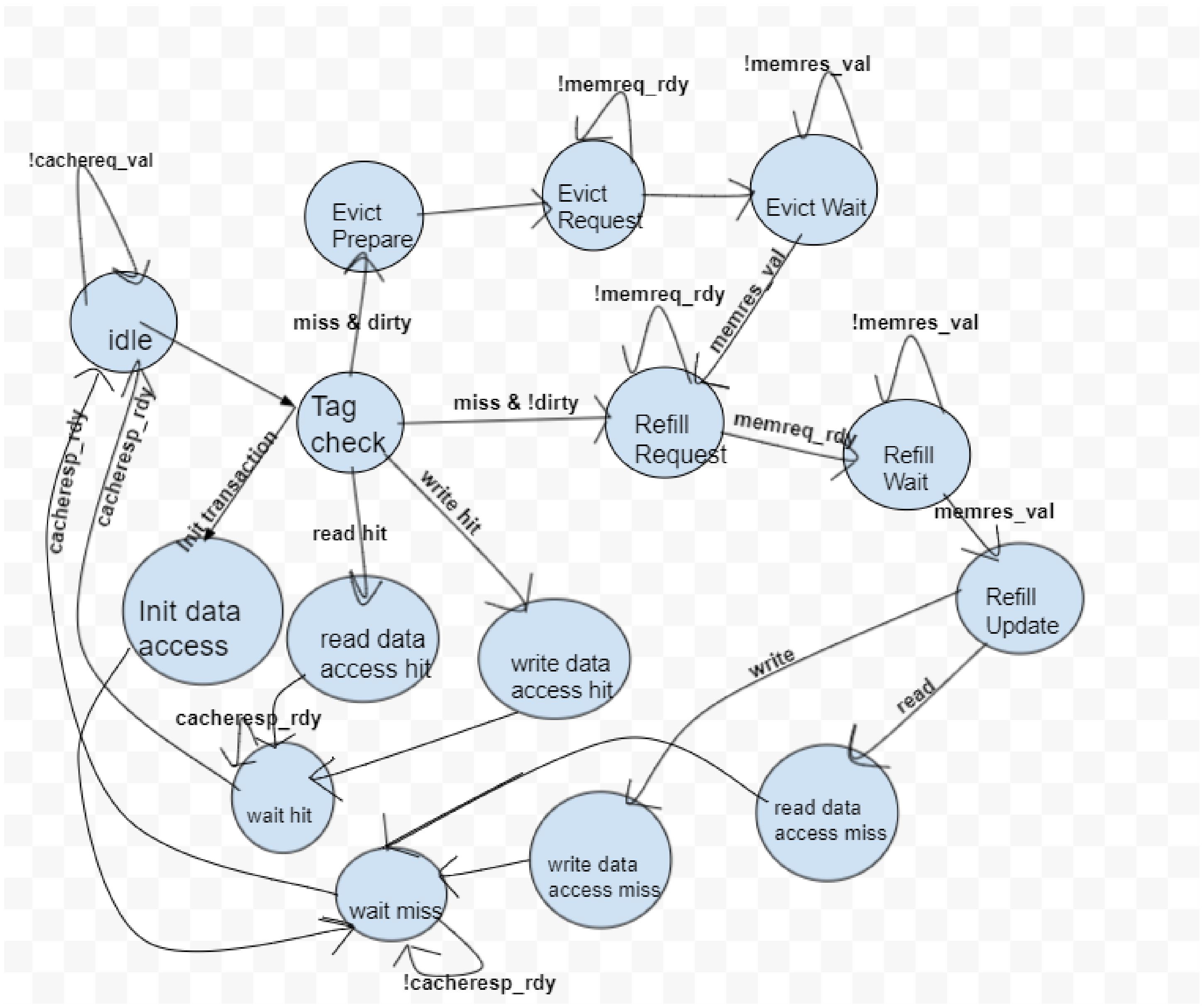

Figure 1: FSM Diagram for the baseline design

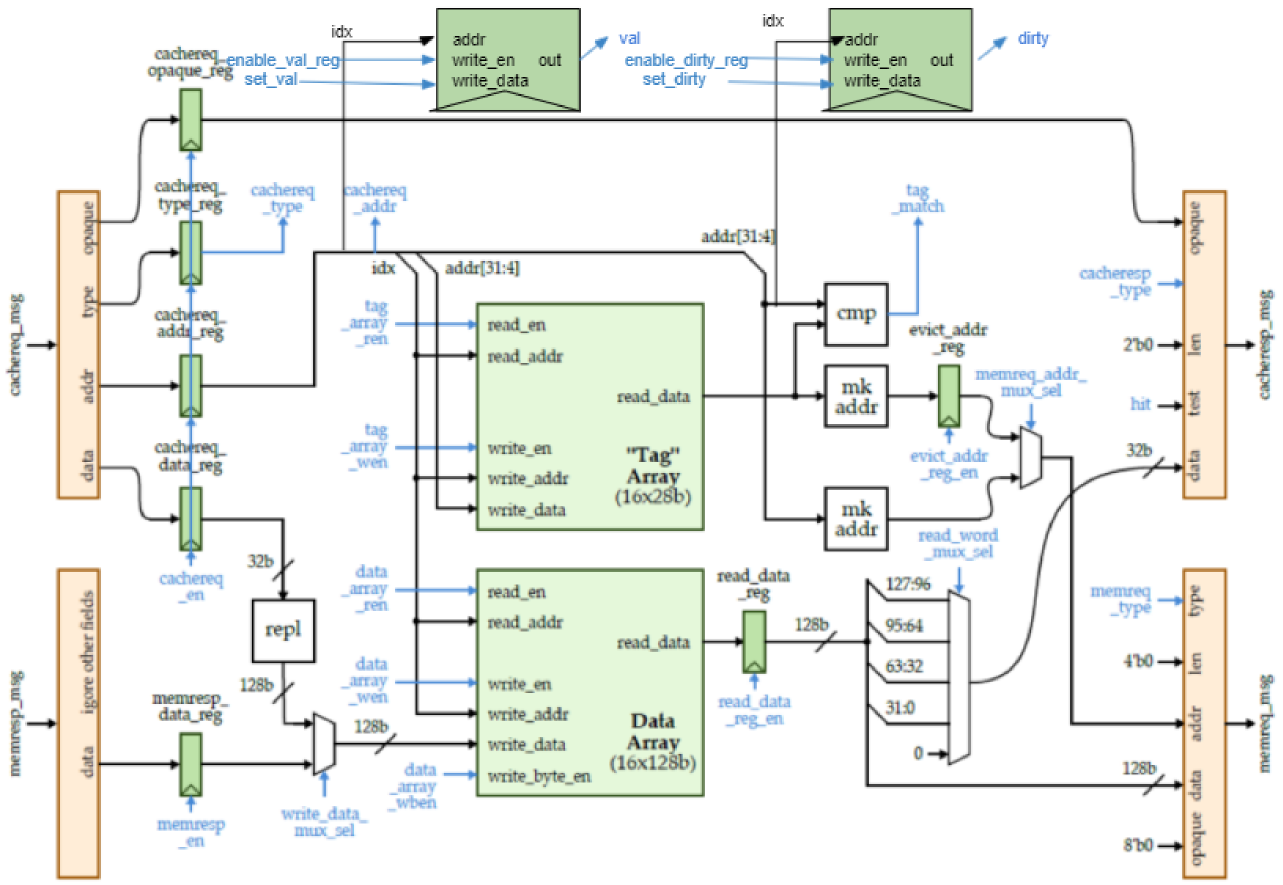

Figure 2: Datapath Diagram for Baseline Design where “repl” means replicate 32b four times to create a 128b signal, Orange blocks represent extracting or inserting fields into a Verilog struct.

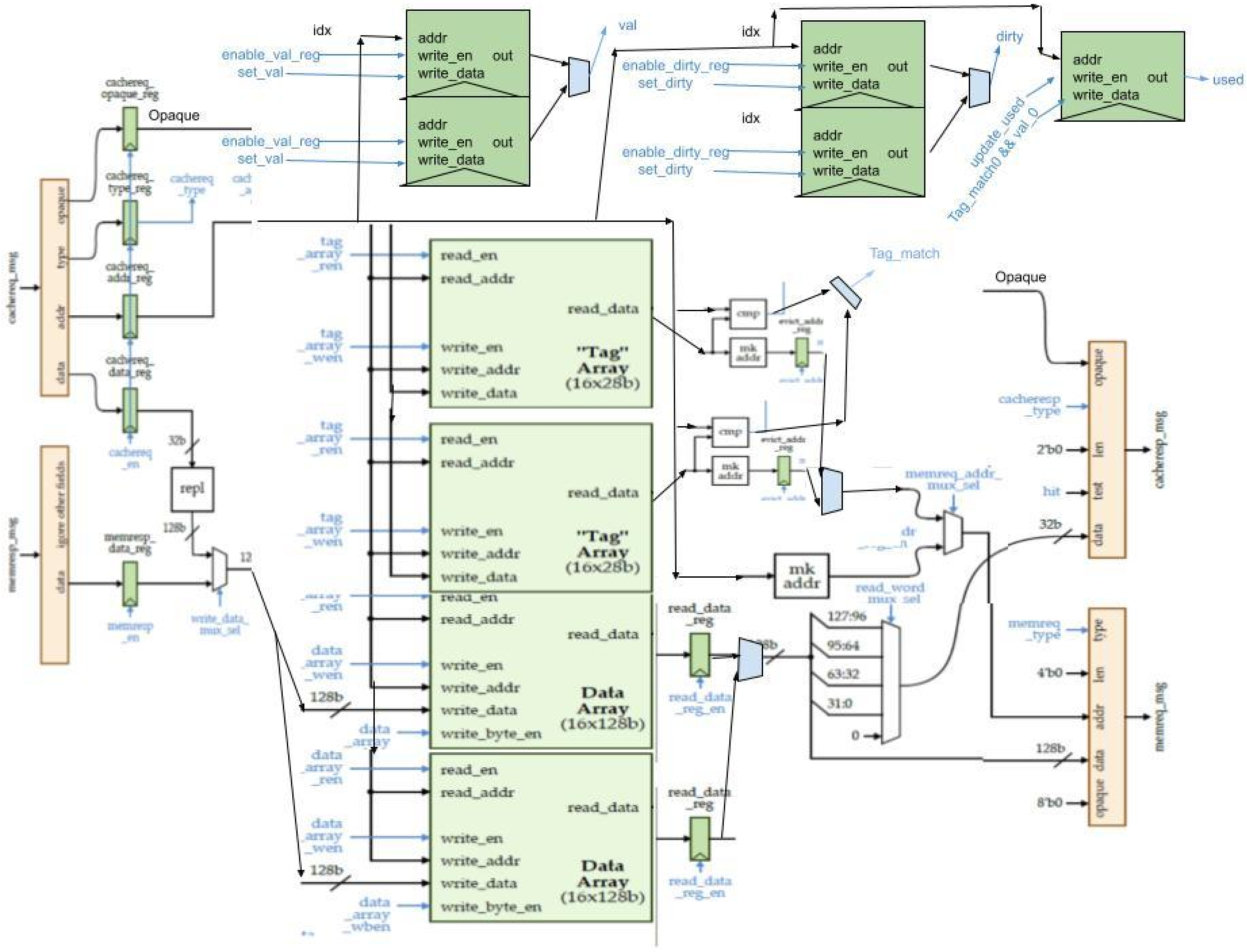

Figure 3: Alternative Datapath

			Baseline Design			Alternative Design		
	num_cycles	num_request	num_misses	miss_rate	amal	num_misses	miss_rate	amal
loop-1d	976	100	25	0.25	9.76	25	0.25	9.76
loop-2d	4232	500	97	0.194	8.464	125	0.25	9.752
loop-3d	2161	80	80	1.0	27.0125	16	0.2	8.6125
vv_add_tmp	7291	270	190	0.7037	27.004	100	0.3704	14.056
vv_add_spac	7291	270	190	0.7037	27.004	132	0.4889	20.870
rand_pat	2708	96	76	0.7917	28.208	84	0.875	34.198
reverse_arr	9213	900	150	0.1667	10.237	160	0.1778	11.489
lin_search	48680	6000	1073	0.1788	8.113	751	0.1252	6.879
read_wr_repeat	867840	56000	14000	0.25	15.497	14	0.0003	4.006

read_miss	811	30	30	1.0	27.033	30	1.0	27.033
read_miss_repeat	2321	120	80	0.6667	19.342	72	0.6	17.808
write_miss	1133	30	30	1.0	37.767	30	1.0	37.767
write_miss_repeat	3793	120	80	0.6667	31.608	72	0.6	28.542

Table 1: A list of evaluation patterns and the resulting hit and miss data